**AFRL-IF-RS-TR-2005-196**
**Final Technical Report**
**May 2005**

# FORGES: FORMAL SYNTHESIS OF GENERATORS FOR EMBEDDED SYSTEMS

**Kestrel Institute**

**Sponsored by**
**Defense Advanced Research Projects Agency**
**DARPA Order No. K217**

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**
**ROME RESEARCH SITE**
**ROME, NEW YORK**

## STINFO FINAL REPORT

APPROVED: /s/
NANCY A. ROBERTS
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Advanced Computing Division
Information Directorate

| **REPORT DOCUMENTATION PAGE** | | | *Form Approved OMB No. 074-0188* |
|---|---|---|---|
| Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503 | | | |
| **1. AGENCY USE ONLY (Leave blank)** | **2. REPORT DATE** May 2005 | **3. REPORT TYPE AND DATES COVERED** **Final** **Jun 00 – Jun 04** | |
| **4. TITLE AND SUBTITLE** FORGES: FORMAL SYNTHESIS OF GENERATORS FOR EMBEDDED SYSTEMS | | | **5. FUNDING NUMBERS** C - F30602-00-C-0155 PE - 62302E PR - MOBI TA - 00 WU - 02 |
| **6. AUTHOR(S)** Lindsay Errington | | | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Kestrel Institute 3260 Hillview Avenue Palo Alto CA 94304 | | | **8. PERFORMING ORGANIZATION REPORT NUMBER** N/A |
| **9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)** Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington VA 22203-1714; AFRL/IFT 525 Brooks Road Rome NY 13441-4505 | | | **10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2005-196 |
| **11. SUPPLEMENTARY NOTES** AFRL Project Engineer: Nancy A. Roberts/IFT/(315) 330-3566 Nancy.Roberts@rl.af.mil | | | |
| **12a. DISTRIBUTION / AVAILABILITY STATEMENT** *APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.* | | | **12b. DISTRIBUTION CODE** |

**13. ABSTRACT *(Maximum 200 Words)***

A number of tools exist that allow engineers to construct models of embedded systems. Models are expressed in a variety of languages including domain specific languages. These models provide input to generators that: 1) produce code, test suites, views of components in the model, and/or 2) analyze or compose models. Generators, however, are often difficult and expensive to develop. Moreover, due to the safety critical nature of embedded systems, it is crucial that generators be high assurance. This project has developed technology for the automated synthesis of model-based generators from language meta-models. Using partial evaluation, Kestrel has demonstrated the synthesis of generators that are provably correct, and that can be produced and modified with drastically less time and effort compared with manual production. The success of the project can be traced to two major contributions. The first, a technology breakthrough, is a new tractable formulation of partial evaluation. The second is a collection of meta-models that serve as comprehensive definitions of the semantics of widely-used commercial modeling languages.

| | | | |
|---|---|---|---|
| **14. SUBJECT TERMS** Model-based generators, partial evaluation, embedded systems model based development, Matlab, Stateflow, high assurance of COTS, production code | | | **15. NUMBER OF PAGES** 41 |
| | | | **16. PRICE CODE** |
| **17. SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED | **18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED | **19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED | **20. LIMITATION OF ABSTRACT** UL |

## Contents

## List of Figures

## 1 Summary

A number of tools exist that allow engineers to construct models of embedded systems. Models are expressed in a variety of languages including domain specific languages. These models provide input to generators that *(i)* produce code, test suites, views of components in the model and/or *(ii)* analyze or compose models. Generators, however, are often difficult and expensive to develop. Moreover, due to the safety critical nature of embedded systems, it is crucial that generators be high assurance.

The FORGES project has developed technology for the automated synthesis of model-based generators from language meta-models. Using partial evaluation, Kestrel has demonstrated the synthesis of generators that are provably correct, and that can be produced and modified with drastically less time and effort compared with manual production.

The success of the project can be traced to two major contributions. The first, a technology breakthrough, is a new tractable formulation of partial evaluation. The second is a collection of meta-models that serve as comprehensive definitions of the semantics of widely-used commercial modeling languages.

## 2 Introduction and Motivation

In recent years there has been an increase in the use of "model-based" languages and associated tools for the development of embedded systems. Examples include the MathWorks suite consisting of Matlab, Simulink and Stateflow as well as languages in the Unified Modeling Language (UML) family including Statecharts and languages like Specification and Description Language (SDL) used primarily in telecommunications. Using one or more of these tools, an engineer can develop, simulate and test a model before targeting it to a specific hardware/software platform. A question that arises is: to what extent can the mapping from model to concrete code be automated?

The same period has also seen the development of a number of analysis tools for embedded systems. This includes tools for schedulability analysis, configuration tools and model checkers. Each of these tools comes with its own modeling language. This leads us to a second question: how can the these tools be integrated in such a way that, for example, an engineer can develop a model in Stateflow [7] and then apply the Spin model checker [3] to verify that the model has or does not have a property of interest.

These two questions represent instances of a common and familiar problem, which is how to translate from one language to another. It has become standard to refer to a tool that does such a translation as a *generator*.

Figure 1 depicts some of the combinations of interest. There are many more. Clearly writing a tool for each instance is not feasible. This is the same problem encountered by compiler writers who address it by introducing a common intermediate language thereby replacing a product of combinations with a sum.

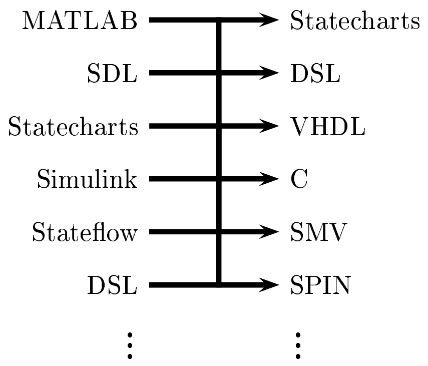


Figure 1: Generator Combinations

At first glance, the task of producing some of these generators, whether through an intermediate language or not, might not seem difficult. Stateflow, for instance, is a state-machine language and intuitively, one might assume that mapping Stateflow to C would be straightforward. However, this is not the case. The apparent visual simplicity is misleading. Languages like Stateflow, Simulink and Statecharts have surprisingly complex and subtle semantics. There are no less than a dozen published semantic definitions for Statecharts.

What complicates matters further is that embedded systems are often safety or financially critical and hence high assurance is essential for all the generators that are used in their development. Obviously, code generators must be correct. But, likewise, there is little value in a generator that connects, for example, Statecharts with a model checker for verification purposes if one does not have a high degree of confidence in the mapping.

Some readers may be puzzled by the implication that a Stateflow to C generator is an interesting problem, bearing in mind that there are at least two commercial vendors that provide such tools. We contend, however, that it remains an interesting problem. Ford, for example, uses Simulink and Stateflow for the development of all their powertrain controllers. Surprisingly, very little auto-generated code is used in production Ford vehicles. Models are developed using Simulink and Stateflow and then translated *by hand* into C. The decoupling of model from code creates two problems. First, all the testing performed on the model must be repeated for the C code. Second, according to Ford, the model and code gradually diverge. When a problem arises, it is all too common for an engineer to simply adjust the code but fail to reflect the change in the model. It is at this point that "the model-based paradigm breaks down" [8]. Precisely the same problem has been expressed to us by a number of groups involved in the development of embedded systems.

There are at least three reasons why auto-generated code is not used. The first is that the quality of the auto-generated code (in terms of readability, code size, *etc.*) is relatively poor. To be sure, vendors have worked hard to improve the quality of their code, nevertheless, it is difficult to come close to the quality of hand-generated code. Earlier we mentioned the subtlety in languages like Stateflow and Statecharts. It is because of this subtlety that organizations like Ford have a "style guide" for Simulink and Stateflow that effectively restricts an engineer to a small subset of the languages. Moreover, engineers are encouraged to use certain idioms in their models. Unlike a programmer, commercial code generators cannot exploit the opportunities that arise by the restrictions to the language and are not sensitive to the idioms used.

The second reason why auto-code generators are rarely used is that the generated code does not conform to the target architecture. It is rare that an embedded system is developed solely inside the model-based paradigm. There may be an off-the-shelf or custom kernel plus drivers and there may be constraints on the structure of the store, placement of variables and interfaces to procedures that are difficult to match by a code generator.

There is an important point in this. We have seen that a code generator, for example from Stateflow to C, must target not only a language but a platform and we have also seen that different users choose to work with different subsets of the languages and use different modelling idioms. The consequence is that there is no *universal* Stateflow to C generator. There are as many generators as there are groups that use the tools. Command line options and menu options will not come close to providing sufficient flexibility. It seems Figure 1 vastly underestimates the scale of the problem.

A third reason why auto-code generators are rarely used is the lack of certified tools for safety critical applications. Often if a tool is integral to the construction of production software it must be certified as correct. Demonstrating a compiler is correct is difficult. If the code generator is not trusted then testing or other V&V activities must be performed on the target code of the generator and not the source.

It is worth pointing out that the need for a *family* of generators for fixed source and target languages applies also when the target is a model checker. Model checkers analyze models having finite state spaces whereas the source languages typically have infinite state spaces. Therefore, an additional task for a generator is to perform an abstraction that takes a model into a finite state space. However, different abstractions are appropriate for different verification tasks and consequently different generators are needed for different abstractions.

Summarizing, to facilitate the development of embedded systems and the integration of embedded systems tools, there is a need for the capability to rapidly develop families of high-assurance generators that produce high-quality specialized code.

### 2.1 Generating Generators

The report proposes a novel approach to developing generators. Rather than write generators, we advocate that they should be *synthesized automatically.* More concretely, generators should be synthesized from *meta-models* for the source and target language. By meta-model for a language we mean a precise and formal specification of the semantics of the language. The remainder of the report describes an approach to synthesizing generators centered around a technique called partial evaluation.

The sections that follow present new mathematical foundations for partial evaluation. This is a breakthrough contribution of the project and a key enabling technolgy for generator synthesis. Earlier work on partial evaluation (see [4]), while promising, has proven to be largely intractable in practice. See [1] for a different approach to the partial evaluation of Matlab.

We also describe our work developing meta-models for Stateflow, Matlab and a domain specific language for software radio. These meta-models represent another major contribution of this project as they standalone as comprehensive reference definitions of the semantics of the respective languages.

## 3 Partial Evaluation

Partial evaluation is a technique for *specializing* a program when some, but not all, inputs are known. The effect is captured in the equations below. Let $P$ be a program with inputs $x$ and $y$ and $S$ be the partial evaluator or specializer. Then:

$$\begin{aligned} P(x,y) &= z \\ S(P, x=t) &= P_{x=t} \\ P_{x=t}(y) &= z \end{aligned}$$

In words, specializing $P$ with $x = t$ yields a program that takes only $y$ as input but returns the same result as $P$. Here $P_{x=t}$ is called the *residual* program. It is a version of $P$ specialized for a specific instance of $x$. These equations also characterize the correctness of partial evaluation. A simple example is shown in Figure 2.

The algorithm on the left computes $x^y$. The algorithm on the right is the *residual program* when the first is specialized for the fixed input $y = 5$. The residual program is obtained by symbolically executing the program and substituting the possible values that $y$ takes on at each point. Context dependent simplification is then applied to reduce expressions and eliminate unreachable code yielding a smaller and more efficient program.

The application of partial evaluation to translation and compilation has been studied extensively in the literature [4]. In this paper we use the simplest instance. With respect to the equations given above, assume $P(x,y)$ is an *interpreter* or *meta-model* for some language $L$ written in another language $M$. Here we assume the argument $x$ is a program in $L$ and $y$ is an input suitable for that

```
proc $exp$ $(x : Nat, y : Nat) : Nat$ = {
let
var $z : Nat$
in
    $z := 1$;
    while $(y > 0)$
        while $even\ y$
            $x := x^2$;
            $y := y$ div 2
        $y := y - 1$;
        $z := z \times x$
    return $z$
}
```

```
proc $exp_{y=5}$ $(x : Nat) : Nat$ = {
    return $x \times ((x^2)^2)$
}
```

Figure 2: Algorithm for computing $x^y$ and its specialization for $y = 5$.

program. Specializing the interpreter with respect to a particular program $t$ in $L$ yields a residual program $P_{x=t}$ in language $M$. Thus, in effect, specialization has translated a program in $L$ into an equivalent program in $M$.

It is important to observe that the result of specialization is a program in the same language as the interpreter.

The equations given earlier are a standard way of presenting the effect of program specialization. Note, however, that they only constrain the value of $P_{x=t}$ and do not uniquely determine what the specializer $S$ actually computes. Indeed, there are many programs that satisfy the constraints on $P_{x=t}$ that one would not normally consider to be specializations of $P$. For instance, rather than an imperative language, consider the $\lambda$-calculus. Let $P = \lambda(x, y) \,.\, M$ for some term $M$. Then the *Curried* term $P_{x=N} = \lambda y \,.\, P\ (N, y)$ for some $x = N$, satisfies the equations but is not an interesting specialization of $P$.

What is needed, therefore, is a precise charactization of the transformation $S$ applied to compute $P_{x=t}$ from $P$. In the literature, this transformation is presented as an algorithm that operates on the abstract syntax of $P$. The problem is that for all but trvial languages, this transformation is subtle and complex, and that complexity creates an obstacle to proving its correctness.

In the next section, we sketch a semantics-based theory of algorthims as flow-graphs. This theory affords us the ability to *specify* the specialization transformation abstractly and in purely mathematical terms, thus making it possible to prove the correctness of an algorithm that implements the transformation.

## 4 Behavioral Specifications

This section introduces *behavioral specifications*, a key innovation in the project.

Intuitively, a behavioral specification or BSpec is a *flow-graph*, *state machine* or *transition system*. More concretely, a BSpec is a labeled graph. The vertices of a graph are states and the edges are transitions. Each vertex is labeled with a set of typed variables. These are the variables in scope at that point in the program. Each edge is labeled with a logical formula. The formula denotes a *guarded command*. It describes both *when* the transition can be taken and how the variables *change* when the transition is taken. More precisely, the formula classifies the relation between the values of the variables at the start of the transition with those at the end. We adopt the convention that primed names refer to the new value of a variable. This yields yields specifications much like those used in Z [9]. See also Lamport's TLA [5].

The most general definition of BSPECS is given using constructions from category theory. The category theory explicates the connection between, on the one hand, familiar operations on flow graphs and machines found in computer science and, on the other hand, standard constructions from category. For this paper, a simpler and less general definition is sufficient.

**Definition 1** *A* BSPEC *is a tuple* $(G, s, F, L)$ *where*

- $G$ *is a directed graph*
- $s$ *is the start vertex*
- $F$ *is a set of final states*
- $L$ *is a labeling function on the states and transitions such that*
  - *for each vertex* $v$*,* $L(v)$ *is a set of typed variable names*
  - *for each edge* $e : v \longrightarrow w$*,* $L(e)$ *is a logical formula,* $\theta$*, where the free variables in* $\theta$ *come from the disjoint sum* $L(v) + L(w)$ *such that* $\theta$ *classifies a relation between the values of the variables at* $v$ *and the variables at* $w$*.*

Figure 3 shows a fragment of code in a context and the corresponding graph and labeling. The graph has three states, $a$, $b$ and $c$, and two transitions, $f$ and $g$. Each vertex is labeled with the names of the two variables in scope and the transitions are labeled with logical formulas that encode the effect of the two steps in the program.

## 4.1 BSPECS and Partial Evaluation

Next we formally define specialization as a transformation on BSPECS.

We begin with some terminology. A *substitution* is a conjunction of equations of the form $x_i = t_i$ where $x_i$ is a variable and $t_i$ is a term. A *constant* is a term consisting solely of numbers, constructors and applications of constructors to constants. For instance, if *nil* and *cons* are the usual constructors for lists, then $cons(1, nil)$ is a constant but $cons(1, x)$ is not. A *ground substitution* is a conjunction of equations of the form $x_i = c_i$ where each $c_i$ is a constant. Given a formula $\theta$ and a substitution $\sigma$, we write $\sigma(\theta)$ for the formula obtained by applying the substitution $\sigma$ to $\theta$. Semantically, $\sigma(\theta) = \sigma \wedge \theta$.

$$\begin{array}{l} x : \mathbf{Nat}, y : \mathbf{Nat} \vdash \\ \quad x := x + 1; \\ \quad \mathsf{if}\ x > y\ \mathsf{then} \\ \qquad x := x + y \end{array}$$

$$\begin{tikzcd}
a \arrow[d, no head] \arrow[r, dotted, maps to] & x, y : \mathbf{Nat} \arrow[d, dotted, no head] \\
e \arrow[d] \arrow[r, dotted, maps to] & x' = x + 1 \arrow[d, dotted] \\
b \arrow[d, no head] \arrow[r, dotted, maps to] & x, y : \mathbf{Nat} \arrow[d, dotted, no head] \\
f \arrow[d] \arrow[r, dotted, maps to] & x > y \wedge x' = x + y \arrow[d, dotted] \\
c \arrow[r, dotted, maps to] & x, y : \mathbf{Nat}
\end{tikzcd}$$

Figure 3: Program fragment in context, graph and labeling

The *postcondition* of a transition is the strongest formula that classifies the values the variables can take on at the end state. Mathematically, given a transition $e : a \longrightarrow b$, labeled $\theta$, the postcondition of $\theta$ is the formula

$$\exists vars(L(a)) \,.\, \theta$$

modulo priming of names. For instance, the postcondition of the second transition in Figure 3 is a formula where only $x$ is free, namely

$$\exists x'', y'' \,.\, x'' > y'' \wedge x = x'' + y''$$

This, of course, simplifies to $\mathsf{true}$. The *ground postcondition* of a transition $\theta$ is the strongest ground substitution implied by the postcondition of the transition. For instance, conjoining the substitution $[x'' = 3 \wedge y'' = 4]$ (a precondition) to the second transition in the example gives a postcondition that simplies to $x = 7$. This is a ground postcondition.

**Definition 2** *Given a* BSPEC $P = (G_P, s_P, F_P, L_P)$ *and a ground substitution, $\sigma$, for some of the variables labeling the initial state, $s_P$, the specialization of $P$ with respect to $\sigma$ is* BSPEC $Q = (G_Q, s_Q, F_Q, L_Q)$ *defined inductively as follows:*

- $s_q = (s_p, \sigma)$
- $L(s_q) = L(s_p) \setminus vars(\sigma)$
- *Let $(a, \sigma)$ be a vertex in $G_Q$, $e : a \longrightarrow b$ be an edge in $G_P$ and $\rho$ be the ground postcondition of $L(e) \wedge \sigma$. Then*
  - $(b, \rho)$ *is vertex in* $G_Q$

– $(e, \sigma, \rho) : (a, \sigma) \longrightarrow (b, \rho)$ *is an edge in* $G_Q$

– $L(b, \rho) = L(b) \setminus vars(\rho)$

– $L(e, \sigma, \rho) = \exists vars(\sigma), vars(\rho) \,.\, L(e)$

– $b \in F_P$ *implies* $(b, \rho) \in F_Q$

$$\begin{tikzcd}
a \arrow[r, "e"] & b \arrow[r, "g"'] \arrow[d, "m"'] & c \arrow[l, bend right, "h"'] \arrow[loop right, "k"] \\
& d &
\end{tikzcd}
\qquad
\begin{tikzcd}
T \arrow[r, "z' = 1"] & T \arrow[r, "y \neq 0"'] \arrow[d, "y = 0"'] & T \arrow[l, bend right, "\begin{array}{l}\neg(2 \mid y) \\ \wedge\, y' = y - 1 \\ \wedge\, z' = z * x\end{array}"'] \arrow[loop right, "\begin{array}{l}2 \mid y \\ \wedge\, x' = x^2 \\ \wedge\, y' = y/2\end{array}"] \\
& T &
\end{tikzcd}$$

$$T = \boxed{x, y, z : \mathbf{Nat}}$$

Figure 4: Graph and corresponding BSpec, $P$, for exponential algorithm.

Figure 4 shows the BSpec, $P$, for the body of the exponential algorithm given earlier. Note that to simplify the figure, some pairs of assignments have been combined into single transitions in the BSpec.

Figure 5 illustrates the residual BSpec, $Q$, obtained by specializing the algorithm in Figure 4 with the ground substitution $[y = 5]$. Figure 4 shows both the new graph and its labeling.

## 5 Meta-modeling in Oscar

BSpecs underlie the theory and implementation of the specializer. However, a goal of the project is that each meta-model should serve not only as input to the specializer, but also as a reference document, made available to an application engineer, that clearly and unambiguously defines the semantics of the language. To that end, a meta-model must be as simple and as abstract as possible and for that task BSpecs are inappropriate.

Instead, we have developed a simple procedural specification language called *Oscar*. Oscar is a hybrid of an algebraic specification language and Dijkstra's guarded command language. The control structures include procedures, variable declarations, assignments, guarded commands and iteration. A compiler transforms each Oscar procedure into a BSpec.

$$\begin{tikzcd}
(a,[y=5]) \arrow[r, "{(e,[y=5],[y=5])}"] & (b,[y=5]) \arrow[r, "{(g,[y=5],[y=5])}"] & (c,[y=5]) \arrow[dl, "{(h,[y=5],[y=4])}"'] \\
 & (b,[y=4]) \arrow[r, "{(g,[y=4],[y=4])}"] & (c,[y=4]) \arrow[d, "{(k,[y=4],[y=2])}"'] \\
 & & (c,[y=2]) \arrow[d, "{(k,[y=2],[y=1])}"'] \\
 & & (c,[y=1]) \arrow[dl, "{(h,[y=1],[y=0])}"'] \\
 & (b,[y=0]) \arrow[d, "{(m,[y=0],[y=0])}"'] & \\
 & (d,[y=0]) &
\end{tikzcd}$$

$$\begin{tikzcd}
T' \arrow[r, "z'=1"] & T' \arrow[r, "\mathbf{true}"] & T' \arrow[dl, "z'=z*x"'] \\
 & T' \arrow[r, "\mathbf{true}"] & T' \arrow[d, "x'=x^2"'] \\
 & & T' \arrow[d, "x'=x^2"'] \\
 & & T' \arrow[dl, "z'=z*x"'] \\
 & T' \arrow[d, "\mathbf{true}"'] & \\
 & T' &
\end{tikzcd}$$

$$T' = \boxed{x, z : \mathbf{Nat}}$$

Figure 5: Specialization of exponential BSpec for $y = 5$.

## 6 Interpretations

As mentioned earlier, the result of specializing an interpreter with respect to a particular program is a residual program in the same language as the interpreter. In our context, this means that the result of partial evaluation is a BSpec. Having explicated the semantics of the input program as a BSpec, the next task is to map that semantic representation to the target language. For this purpose we use the notion of an *interpretation*.

We should mention that there is an alternative approach where the interpretation component is omitted. If, for example, the goal is to generate C, then one could develop a specializer for C and write the interpreter/meta-model in C. In this way, applying the partial evaluator would translate a model from the source language directly to C. This option was rejected for two reasons. First, C is a low-level language, and therefore, like BSpecs, inappropriate as a meta-modeling language.

The second reason for not writing a specializer for a language like C stems from the complexity of the language. It would be a difficult task to write a specializer capable of performing all the necessary simplifications and inferences.

A lesson from this project is that higher abstraction simplifies specialization. For instance, finite sets and maps are used extensively in meta-models. At an abstract level, each of these types is defined by a theory with three or four operations and a handful of axioms. Reasoning about such types is straightforward. By contrast, if one implements finite maps and sets in C, then the reasoning becomes much more complex, as it involves state, the heap and pointers.

## 7 Meta-Models

### 7.1 Stateflow

Stateflow is a state machine modeling tool, similar to Statecharts [2], and integrated into MATLAB's Simulink [6] tool set. While the inspiration for Stateflow is based on the simple and intuitively clear behavior of state machines, interacting features and extensions have led to complex semantics.

**State hierarchy** States are organized as a tree structure. The substates of a state may decomposed in parallel or sequentially. When a parallel decomposed state is entered (exited) all of the substates are entered (exited); for sequential states only one substate is entered. The semantics of stateflow is completely sequential and deterministic, based on complex rules ordering entrance and exiting of states, and testing of transitions.

**Junctions** Transitions segments are edges between two types of nodes: states and junctions. A transition is a path from a state to a state passing through junctions. A backtracking search is used to identify the next transition to traverse.

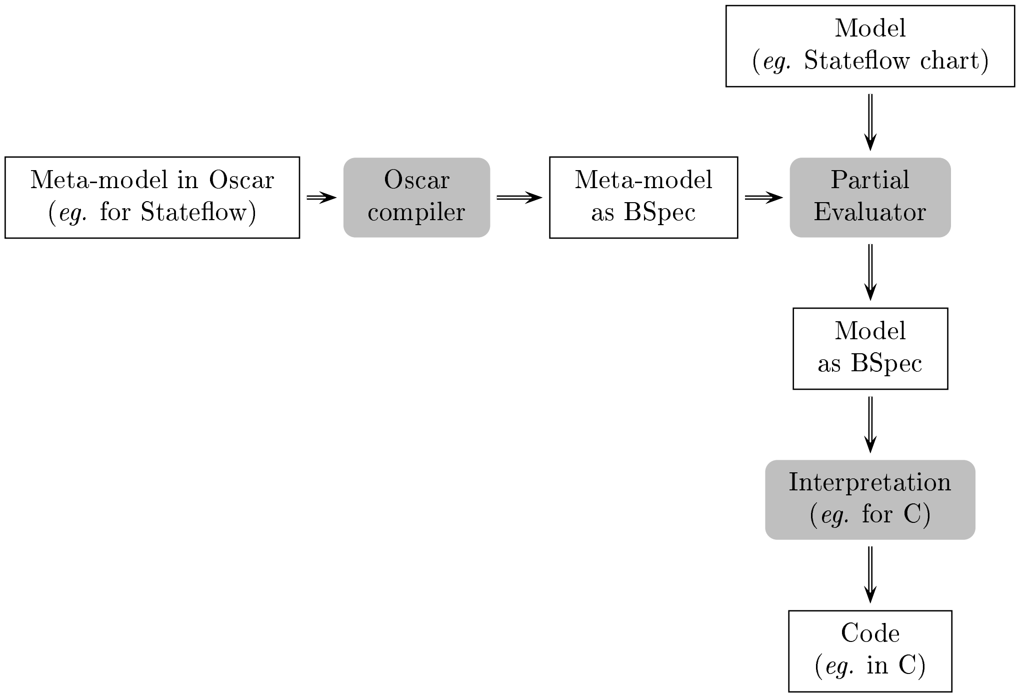


Figure 6: Structure of a generator.

**Data Model** Events and variables may be declared and scoped within a state. Actions may be associated with the execution of transitions, entering/leaving states, and include updating the values of variables, and invocation of events. Stateflow supports a variety of types and operations some loosely based on C others inherited from Simulink.

**Recursive event invocation and early return** Execution is event based. A step of the Stateflow machine is initiated by an external event. However execution of an action may broadcast a new event which leads to recursive response to the event. Upon completion of the recursion, processing of the pending event may or may not terminate.

Despite this complexity, Stateflow is widely used, but often organizations restrict its use to a more intuitive and manageable subset. The semantics are described in a reference manual, often relying on example. The MathWorks implementation of Stateflow is considered the final arbiter of the semantics, but users may be unaware that their interpretation differs from the MathWorks implementation. We believe that our specification is an accurate, precise, and understandable description of Stateflow semantics.

### 7.1.1 Static Semantics

The semantics of Stateflow is in two parts. The *static semantics* deals with the representation of Stateflow charts used in the meta-model and the conditions for a chart to be well-formed. Here we sketch aspects of the static semantics.

A model consists of a collection of machines, charts, states, etc. In the *.mdl* file representation of a chart, each element is given a unique integer identifier. We use Oscar's predicative subtypes to define unique types for the various sorts of elements in a model. In particular, we define the following collection of predicates:

```
type Id = Nat
op isStateflowNode?         : Id -> Boolean
op isMachineChartStateNode? : Id -> Boolean
op isMachine?               : Id -> Boolean
op isChartStateNode?        : Id -> Boolean
op isChart?                 : Id -> Boolean
op isState?                 : Id -> Boolean
op isEvent?                 : Id -> Boolean
op isJunction?              : Id -> Boolean
op isTransition?            : Id -> Boolean
op isData?                  : Id -> Boolean
```

In the static semantics, these functions are abstract in the sense that their definitions are omitted. The definitions are given later when the *.mdl* file for a particular chart is parsed.

With the predicates, we can define the subtypes.

```
type Id = Nat
type StateflowNode         = (Id | isStateflowNode?)
type MachineChartStateNode = (Id | isMachineChartStateNode?)
type Machine               = (Id | isMachine?)
type ChartStateNode        = (Id | isChartStateNode?)
type Chart                 = (Id | isChart?)
type State                 = (Id | isState?)
type Event                 = (Id | isEvent?)
type Junction              = (Id | isJunction?)
type Transition            = (Id | isTransition?)
type Data                  = (Id | isData?)
```

Charts and states may have *and* and *or* children. For this we define an Oscar *disjoint sum* type and two functions. One assigns children to a chart or state and the other gives the parent of a state.

```
type Children = | Or (List State)
                | And (List State)
                | NoChildren

op children      : ChartStateNode -> Children
op parentOfState : State -> ChartStateNode
```

Here, `Or`, `And` and `NoChildren` are the *tags* or *constructors* of the sum type. Note, once again, that the functions `children` and `parentOfState` are abstract. They are given definitions according to the chart being interpreted.

Transitions connect states and junctions. The destination of a transition may be a state or a junction. Likewise, the source, if it exists, may be a state or a junction.

```
type Destination = | State    State
                   | Junction Junction

op sourceOfTransition      : Transition -> Option Destination
op destinationOfTransition : Transition -> Destination
```

Transitions are labelled with up to four attributes.

| Label field | Description |
|---|---|
| event | Event that causes the transition to be evaluated or fired. |
| condition | A predicate that must be true for the condition action and transition to take place |
| condition_action | If the condition is true, the action specified executes and completes. |
| transition_action | After a valid destination is found and the transition is taken, this action executes and completes. |

We define the corresponding abstract functions.

```
op eventOfTransition           : Transition -> Option Event
op conditionOfTransition       : Transition -> Option Node
op conditionActionOfTransition : Transition -> Option Node
op actionOfTransition          : Transition -> Option Node
```

Transitions are classified in one of three ways. Default transitions have no start state. Inner transitions do not leave the scope of originating state and whereas outer transitions do.

```
op defaultTransitions       : ChartStateNode -> List Transition
op innerTransitionsOfState  : State -> List Transition
op outerTransitionsOfState  : State -> List Transition
```

The remainder of the static semantics follows the same pattern and deals with attributes such as the data in the chart, the events that activate the chart etc.

### 7.1.2 Dynamic Semantics

The other part of the Stateflow meta-model is the *dynamic semantics*. This is a collection of Oscar procedures that collectively define an abstract interpreter for Stateflow.

The execution of a Stateflow program is event driven. Informally, there is an "interpreter" or "machine" that sits waiting for an event. When an event happens, the interpreter seeks a transition waiting for that event where the transition originates from an *active* state. The transition may have a guard associated with it. If the guard is absent or evaluates to true, an optional action associated with the transition is activated. The machine advances to the destination state by setting the source state inactive and the target active.

The behaviour of the interpreter is complicated by a number of factors. We list some below.

1. States are hierarchical. When an event happens the interpreter starts from the root of the hierarchy and moves inwards.

2. Actions may be associated with both states and transitions.

3. There can be not just one, but two actions associated with a transition. One is executed when the condition evaluates to true. The other is executed when the transition is "valid".

4. Or states typically include junction nodes. The transition from one state to another cannot be taken until all conditions for the segments from the source state through the junctions to the destination state are valid. If one discovers in the last segment that the transition cannot take place, Stateflow backtracks looking for another path. Along the way, condition actions may have been executed.

5. There may be transitions at different levels and between levels of the hierarchy.

6. The execution of an event and the transition from one state to another can be premempted if an action for a transition is itself an event broadcast. In such circumstances, the processing of the first event is suspended until processing of the second is complete.

The Oscar procedures defining the dynamic semantics for Stateflow correspond roughly with the various paragraphs in the section of the Stateflow User's Guide titled "Semantic Rules Summary for Stateflow". That is, there are procedures for entering a chart, executing an active chart, entering a state, executing an active state, exiting a state, executing a set of flow graphs etc. This correspondence is an important aspect of the Kestrel approach.

Figure 7 defines the Oscar procedure for exiting a state. It serves to illustrate a number of Oscar constructs. In particular, lines 7 to 18 give a case construct. Note that the guard of an alternative may bind a value to a variable. For instance, the guard `| var (andStates:List State) And andStates ->` is satisfied when `(children state)` is a list of *and* states in which case the variable `andStates` becomes bound within that alternative.

A second fragment of the Oscar semantics for Stateflow appears in Figure 8.

#### 7.1.3 Validating the meta-model

A key step to ensuring the correctness of a generator is validating the meta-model upon which the generator is based. For each of our Stateflow meta-models, this is achieved by comparing the behavior of a representative collection of charts when run in the MathWorks environment with their behavior when run under the interpreter.

The comparison is made by running MATLAB in parallel with a second process that runs the meta-model instantiated with a chart. A *Stateflow proxy* S-function has been written in C. The S-function is added to the Simulink diagram in such a way that the input signals to the chart are copied to the proxy (see Figure 10). When activated, the proxy collects data from its input ports and sends it through a socket to the process running the interpreter. The interpreter executes the chart with the given data, collects the output and sends it back to the proxy. This is illustrated in Figure 9. The output signals from the chart and the proxy are then compared within Matlab by plotting them side-by-side.

## 7.2 MATLAB

The MATLAB to C code generator was motivated by the need to automate the implementation of signal processing applications from MATLAB models at SwRI, a contractor providing an Open Experimental Platform for the MoBIES project.

SwRI developed an analytical framework and implementation architecture for the development of reconfigurable signal classification systems. These sys-

```
proc exitState (state : ChartStateNode, event : Event): ReturnValue
   let
      var rv: ReturnValue
      var ebrc: EBRCondition
      var childState : State
   in
      case (children state)
         | NoChildren -> skip
         | var (andStates:List State) And andStates ->
              do childState in (reverse andStates)
                 rv := exitState (childState, event)
                 if ~(isOK? rv) -> return rv
         | var (orStates : List State) Or orStates ->
              do childState in (reverse orStates)
                 if isActive? childState ->
                    rv := exitState (childState, event)
                    if ~(isOK? rv) -> return rv
                    break
      case (exitActionOfState state)
         | var (node:Node) Some n ->
            ebrc := (true, state)
            rv := execute (node, event, ebrc, state)
            if ~(isOK? rv) -> return rv
      assign (stateName state, mkBool false)
      return OK
```

The execution steps for exiting a state are as follows:

- If this is a parallel state, and one of its sibling states was entered before this state, exit the siblings starting with the last-entered and progressing in reverse order to the first-entered.
- If there are any active children, perform the exit steps on these states in the reverse order they were entered.
- Perform any exit actions.
- Mark the state as inactive.

Figure 7: Semantics of Exit State

```
proc executeActiveState (state : State, event : Event): ReturnValue {
   let
      var result : ReturnValue
      var childState : State
   in
      if ~((outerTransitionsOfState state) = []) ->
         result := executeFlowGraphs(state, outerTransitionsOfState state, event)
         if ~(isFail? result) -> return result
      result := executeDuringAndOnActions (state, event)
      if ~(isOK? result) -> return result
      if ~((innerTransitionsOfState state) = []) ->
         result := executeFlowGraphs(state, innerTransitionsOfState state, event)
         if ~(isFail? result) -> return result
      case (children state)
         | NoChildren -> return OK
         | var (andStates:List State) And andStates ->
            do childState in andStates
               result := executeActiveState (childState, event)
               if ~(isOK? result) -> return result
            return OK
         | var (orStates : List State) Or orStates ->
            do childState in orStates
               if
                  | ~(isActive? childState) -> continue
                  | isActive? childState ->
                     result := executeActiveState (childState, event)
                     return result
            abort ("Active OR State: " ++ (name state) ++ " had no active child")
```

The execution steps for executing an active state are as follows:

- The set of outer flow graphs is executed (see Executing a Set of Flow Graphs). If this causes a state transition, execution stops. (Note that this step is never required for parallel states.)
- During actions and valid on-event actions are performed.
- The set of inner flow graphs is executed. If this does not cause a state transition, the active children are executed, starting at step 1. Parallel states are executed in the same order that they are entered.

Figure 8: Semantics of Execute Active State

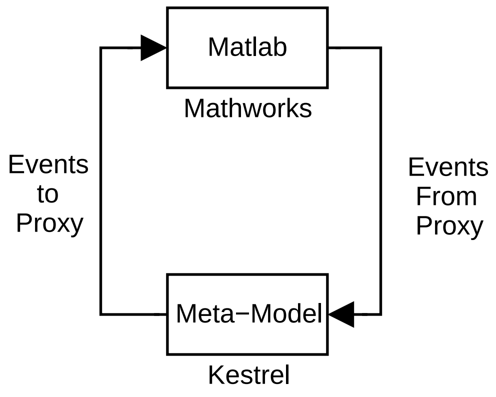
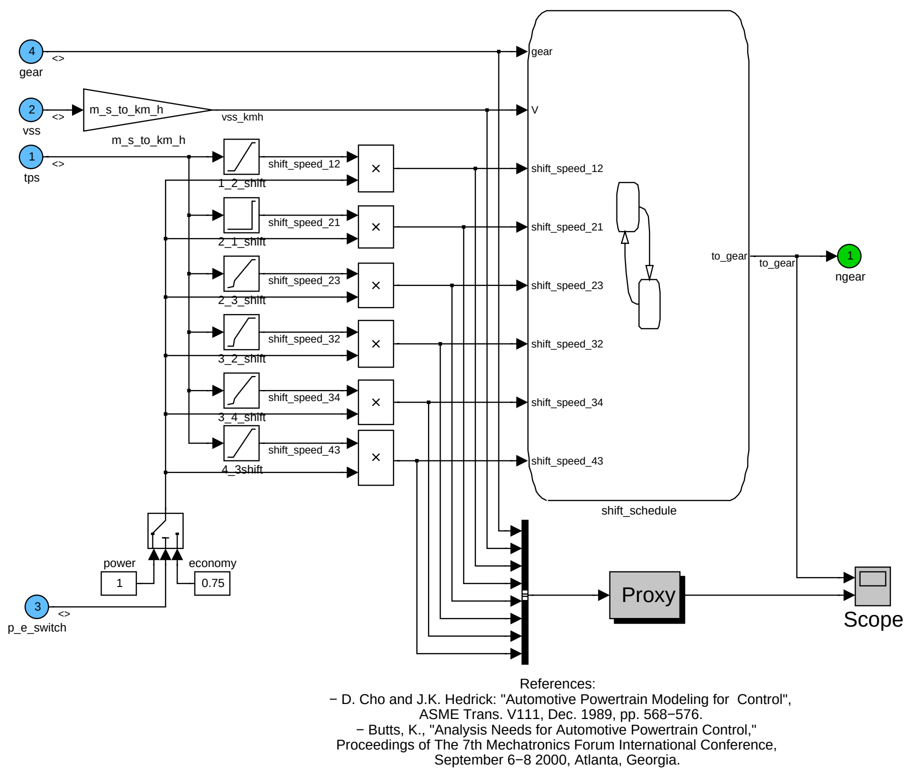


Figure 9: Validating the Stateflow Semantics



References:
− D. Cho and J.K. Hedrick: "Automotive Powertrain Modeling for Control",
ASME Trans. V111, Dec. 1989, pp. 568−576.
− Butts, K., "Analysis Needs for Automotive Powertrain Control,"
Proceedings of The 7th Mechatronics Forum International Conference,
September 6−8 2000, Atlanta, Georgia.

Figure 10: Simulink diagram instrumented with Stateflow S-function proxy

tems are used to autonomously detect incoming signals of interest in radio communication traffic. The analytical framework involves signal processing *opblocks* that are combined to build feature extractors to analyze incoming signals, and *classifier* blocks that are combined to build classification rules that partition the feature space into classification regions. The implementation architecture is closely aligned with the analytical framework, consisting of software components that are combined to implement the feature extractors and classification rules, and a feature scheduler that selects the next feature extractor to run and initiates its execution.

Kestrel's MATLAB to C code generator automates the implementation of the signal processing blocks from the MATLAB models used in the analysis. The code generator is based on the partial evaluation of a meta-model of the MATLAB language, in the same way as the code generator for Stateflow.

The SwRI signal classifier models include information about the data types and desired function signatures of the signal processing blocks. This information is also extracted from the models and used by the code generator.

### 7.2.1 The MATLAB language

MATLAB is a system for technical computing that integrates interactive computation, visualization, and programming. Over the years, the MATLAB language has evolved to support many applications through domain-specific libraries, or *toolboxes*.

The basic data element in MATLAB is an array that does not require dimensioning. There are no variable declarations, and function definitions do not include parameter types. For example:

```
function [x, y] = f(a, b)
  x = sum(a);
  y = x + b;
end
```

If `a` and `b` are arrays with dimensions $m_1 \times n_1$ and $m_2 \times n_2$, with $m_1 > 1$, then `x` will be of dimensions $1 \times n_1$ and `y` will have dimensions $1 \times n_2$. The function will fail unless $m_2 = 1$ and either $n_2 = n_1$ or $n_2 = 1$, but that is not expressed in the code.

Array variables in MATLAB share the same name space with built-in functions, and it is possible to redefine built-in functions locally as array variables. For instance, the built-in `i` is the complex unit, but it does not conflict with a loop variable `i` in a program, as shown below:

```
x = 3 + 2 * i;      // complex unit
for i = 1:10
  x = (x + i).^2;  // loop variable
end
x = x - i;          // i = 10
clear i;
x = i;              // complex unit
```

Array variables may also change type as new data are assigned to them. For instance:

```
x(1) = 2;          // 1x1, double
x(2) = i;          // 1x2, complex (double)
x(2,1) = 3;        // 2x2, complex (double)
```

The features of the MATLAB language facilitate rapid prototyping and the writing of generic and intuitive programs. On the other hand, they require the runtime environment to support dynamic types and dynamic memory management.

### 7.2.2 MATLAB meta-model

The meta-model developed for MATLAB is designed to capture those features that are relevant to SwRI's signal processing blocks. Some features of the meta-model are the following:

**Default base types** Array variables in MATLAB may have different base types. The meta-model includes default base types such as `Real`, `Int`, `Complex`.

**External base types** Base types can be extended through parameterized *external* base types, with specified storage size, alignment, and a mapping to a name in the target language (in this case, C). The default `Real`, `Int`, `Complex` base types are also associated with specific external types in the meta-model. The extensibility of the base type allows data types to be imported from SwRI's signal classifier models. The result of arithmetic operations is always converted to one of the default types.

**Arrays and dimensions** One of the most extensive parts of the meta-model is a set of operations to calculate the base type and dimensions of a MATLAB expression. These operations depend on the expression and on global variables representing the current bindings of variable and function names in the interpreter.

**Bindings** The meta-model contains global lists of bindings of names to addresses, and of addresses to reference types. These lists are initialized with the names, addresses and types of built-in MATLAB functions. Names that are bound to variables during interpretation are added to the head of the list, so the statement `i = 1` will allocate memory for an integer variable, add the bindings of `i` to the allocated address, and of the address to the reference type (array, base type integer, dimensions $1 \times 1$), making the built-in complex unit operator `i` inaccessible until the variable `i` is cleared.[1]

**Function classes** Functions are added to the bindings list with a reference type that allows the type of the return value(s) to be calculated from

[1] The complex unit is a built-in function and not a constant, because it cannot be cleared.

the function arguments. The meta-model includes reference types for the following function classes:

- *Pointwise* functions, with a return type that has the same dimensions as the arguments, and the same or a different base type—e.g. `y = abs(x)`, where a complex array `x` produces a real array `y` with the same dimensions.
- *Constant* functions, with a constant return type—e.g. `y = i`, where the return type is always the same for the built-in function `i`.
- *Strength reductions*, where the return type is reduced along the first non-trivial dimension—e.g. `y = sum(x)`, where a real $2 \times 3$ array `x` produces a real $1 \times 3$ array `y`.
- *Special matrices*, such as `ones(3)` or `zeros(5,6)`, where the type is a function of the input arguments (real with dimensions $3 \times 3$ and $5 \times 6$ respectively).

The function class must be given for each function (built-in or external) that is added to the bindings lists. The function class is used to select the operation in the meta-model that calculates the base type and dimensions of the function's return value(s) from its arguments. If required, the meta-model can be extended with new classes to account for new and specialized functions (e.g. a *histogram* class for the MATLAB built-in `hist` function and all its variations, so `hist(x)` returns a $1 \times 10$ array, `hist(x,n)` returns a $1 \times n$ array, etc.).

**Memory model** The meta-model includes an internal API for memory management in the form of `Alloc` and `Free` statements that the interpreter generates as required. The current meta-model allocates memory for new variables on a private stack, and their addresses are their positions on that stack. External variables (such as function parameters and return values) are assumed to be preallocated, and their addresses are handles to the corresponding variables in the target language. If required, the internal API allows other memory models to extend or replace the private stack in the meta-model.

**Interpreter loop** The main body of the meta-model is a loop that processes the first statement in the program, and either interprets it or replaces it with a simpler statement or sequence of statements to be interpreted in the next pass. This applies to control structures (`for` $\rightarrow$ `while` $\rightarrow$ `if`), memory allocation (assignment to new variable $\rightarrow$ allocation + assignment), etc.

**Expression simplification** Assignment statements that require memory allocation for intermediary results during the evaluation of the right-hand side are replaced with a sequence of assignments to temporary variables, followed by a statement where the intermediary results are replaced by references to the temporary variables. For example, if all variables are $1 \times n$ arrays, then the assignment

```
y = a .* b .* c .* d + e .* f .* g .* h;
```

will be replaced by

```
temp1 = a .* b .* c .* d;
temp2 = e .* f .* g .* h;
y = temp1 + temp2;
clear temp2;
clear temp1;
```

The assignments to `temp1` and `temp2` will be prepended with memory allocation statements and then further simplified in subsequent iterations.

**Type inference** The type of each variable in the program is inferred at the time the variable first appears on the left-hand side of an assignment. The types of variables that appear in a function header (argument or return value) cannot be inferred and must be provided separately in a *types file* provided for each function.[2]

**C code generation** The generation of C code from the interpretation of a model in OSCAR requires an additional translation (or interpretation) step to go from OSCAR to C. This step includes C-specific transformations such as (1) the generation of calls to C functions external to the meta-model and (2) the generation of references to C variables (automatic, or allocated on the private stack) from the variables in the model.

### 7.2.3 Validating the meta-model

The validation of the MATLAB meta-model was performed by applying it to generate C code from the MATLAB opblocks in SwRI's Ethereal Sting experiment E4. The generated opblocks were then used to replace SwRI's C opblocks in a signal classification system and to process a representative set of signals provided by SwRI.

SwRI's signal processing *opblocks* are comparable to Simulink's Embedded MATLAB Function blocks. These blocks support a subset of the language for which the MathWorks tools can generate efficient embeddable code.

The advantage of Kestrel's approach is flexibility. MathWorks tools support a limited set of features that cannot be easily extended by the user: for instance, the MathWorks tools do not allow a variable to be redefined with a different type within a function (which is more restrictive than Kestrel's current meta-model). The set of features supported by the Kestrel tools can be extended by changing the meta-model—e.g., adding new function classes such as *histogram*, or new memory models to match the user's preferred coding style.

[2]The information in the types files is also available in the SwRI models, and may be extracted automatically in the future.

### 7.2.4 Lessons learned

One lesson learned in the development of the MATLAB meta-model is that it may be more effective to specify in advance the subset of the language that will be supported in the meta-model; application-specific meta-models can be made more complete and can generate more efficient code.

Another lesson learned is that most features should be modeled in the higher-level interpreter, independently of the target language, to maximize flexibility and reuse and to minimize translation steps. For instance, one feature that proved problematic was to decide which variables should be assigned `integer` or `double` base types in C. A simpler solution would be to model exactly the semantics of MATLAB, by making them all `double` and making them compare equal when their absolute difference is smaller than `eps`.[3]

## 7.3 SCRAM

The third meta-model is for a language that has become known as the "Software Controlled Radio Abstract Machine" or SCRAM.

The SCRAM is the other half of the analytical framework developed by SwRI for describing signal classification systems. As discussed in the section on the MATLAB meta-model, a signal classifier is a data-flow algorithm. Using graphical editing tools developed by other MoBIES participants, a user draws a data-flow diagram by connecting instances of standard blocks. For instance, there are basic blocks for computing FFTs, peak detection, Kurtosis etc. The behavior of each block is defined both by a MATLAB function, for testing, and by a C++ class, for execution on the target platform. SwRI developed a runtime system for executing the data-flow graphs on the target platform. The runtime system takes an XML description of the data-flow diagram as input, statically determines an execution order for the diagram, creates instances of the classes for the blocks, allocates buffers and then activates the code for each instance according to the execution order.

The SCRAM represents an alternative to the SwRI runtime. It defines an abstract machine for executing the flow graphs. Using a meta-model for the SCRAM we have synthesized a generator that takes the XML representation of the data-flow diagram for a signal analyzer and generates a C program that allocates buffers between the block instances and calls the blocks in an order consistent with the data-flow.

## 8 Assessment

There are a number of metrics for assessing the effectiveness of the approach.

[3]The semantics of MATLAB 7 has made support for integer arithmetic less problematic: in MATLAB 6, if `i` is an integer variable, then `i = i + 1` would result in a double; in MATLAB 7, all terms on the right-hand side are rounded in expressions involving integer variables, so `i = i + 1` will result in an integer (and so will as `i = i + 0.5`, contrary to C intuition).

- Code quality
- Time to develop meta-models

With respect to quality of the code produced by the Stateflow to C generator, according to the midterm evaluations performed by Bill Milam at Ford Research, the approach outlined above produced code that is typically more readable and has a footprint that is comparable or smaller than code generated by commercial tools available from either MathWorks or dSpace. The following quote appears in Bill Milam's midterm report to DARPA:

> The surprising result for us and Kestrel was the quality and size of the code generated. It has taken both dSpace and the MathWorks many years to develop their respective code generation tools. Kestrel took less than two years. In addition, because it is based on an analytic approach to generating the code generator, it is relatively easy to extend the supported Stateflow language and create a new code generator. We believe this approach is extremely promising and hope that commercial tool vendors will take notice.

At the time of writing, we are not aware of the results from the final evaluation, but since the midterm, the partial evaluator has been made 1000× faster than the initial prototype. It has also been improved to eliminate dead code and generate more structured C than the prototype. It is reasonable to assume that a comparison with commercial tools would favor the approach outlined in this report.

With respect to the Matlab to C generator, Denise Varner, an Institute Scientist at SwRI and the OEP leader reports that:

> ... the Kestrel-generated C code produced exactly the same numerical results as the our handwritten C code for 22 out of 45 opblocks in experiment E4. We considered this result very positive, since it reduces the effort needed to code and maintain the C opblocks by 50%. The remaining blocks used features not supported in the Kestrel meta-model, such as cell arrays or Matlab specific graphical operations, or used Matlab built-in functions which could later be replaced by public domain libraries. Simple restrictions on Matlab style and supporting libraries would give essentially complete coverge.

Rather than compare the quality of an individual generator with that of a third party, arguably a more important criterion for assessing the approach outlined in this report is to compare the effort required to produce new generators. It is worth pointing out that the simplest of the three Stateflow meta-models was prepared and validated in less than three person-weeks.

According to standard software metrics, the effort to produce an application is proportional to its size measured as a number of lines. The most comprehensive of the Stateflow meta-models developed in the project has less than 1500 lines. Understandably, vendors do not provide statistics on the size of their tools or on their maintenance costs, but one would be surprised if a handwritten Stateflow to C generator would have fewer than $10x$ that number of lines, and based on our experience writing compilers, the multiplier is probably closer to 100. Even with the more conservative figure, the payoff exceeds a 10-fold improvement.

Further evidence supporting the approach outlined in this report is provided by an experiment performed by Paul Griffiths at UC Berkeley, a member of the MoBIES Automotive OEP team.

Mr. Griffiths used Kestrel's Stateflow to C generator and observed that the code didn't behave the way he expected. The values of some variables were not changing when he thought they should. It boiled down to understanding Stateflow *during* actions. These are actions associated with Stateflow states and executed when Stateflow fails to find a valid transition leaving that state. Mr. Griffiths was under the impression that the action executed whether or not a transition was taken. Comparing the output of the generated code with a simulation inside MATLAB confirmed that former interpretation was correct.

Mr. Griffiths was then directed to the procedure in our semantic definition of Stateflow dealing with *during* actions. As discussed earlier, the definition is a formal transcription of the text in the "Semantics" section of the Stateflow manual. Hierarchies are omitted in the subset we are working with at present.

The semantics contains the fragment:

```
test := executeFlowGraph(outer state);
if ~test -> executeAction (duringAction state)
```

Mr. Griffiths then changed the semantics by omitting the conditional so that *during* actions would be executed whether or not a transition is taken.

He then ran the generator again, using the same chart but with the modified meta-model. This yielded C code for his chart that behaved as he had understood the semantics to be.

It is interesting that the semantics of Stateflow wasn't clear even to an experienced user. Also, while we are not advocating that one change the semantics of Stateflow, the experiment demonstrates how a meta-model can be modified to yield different generators. It also demonstrates the insidious problems that can arise for V&V when Stateflow or Matlab is provided as a model for manual development of C code.

## A Stateflow Example

Figure 11 shows a Stateflow diagram taken from the models developed by the automotive OEP. It defines a transmission shift scheduler. The remainder of this appendix gives the Kestrel generated C code.

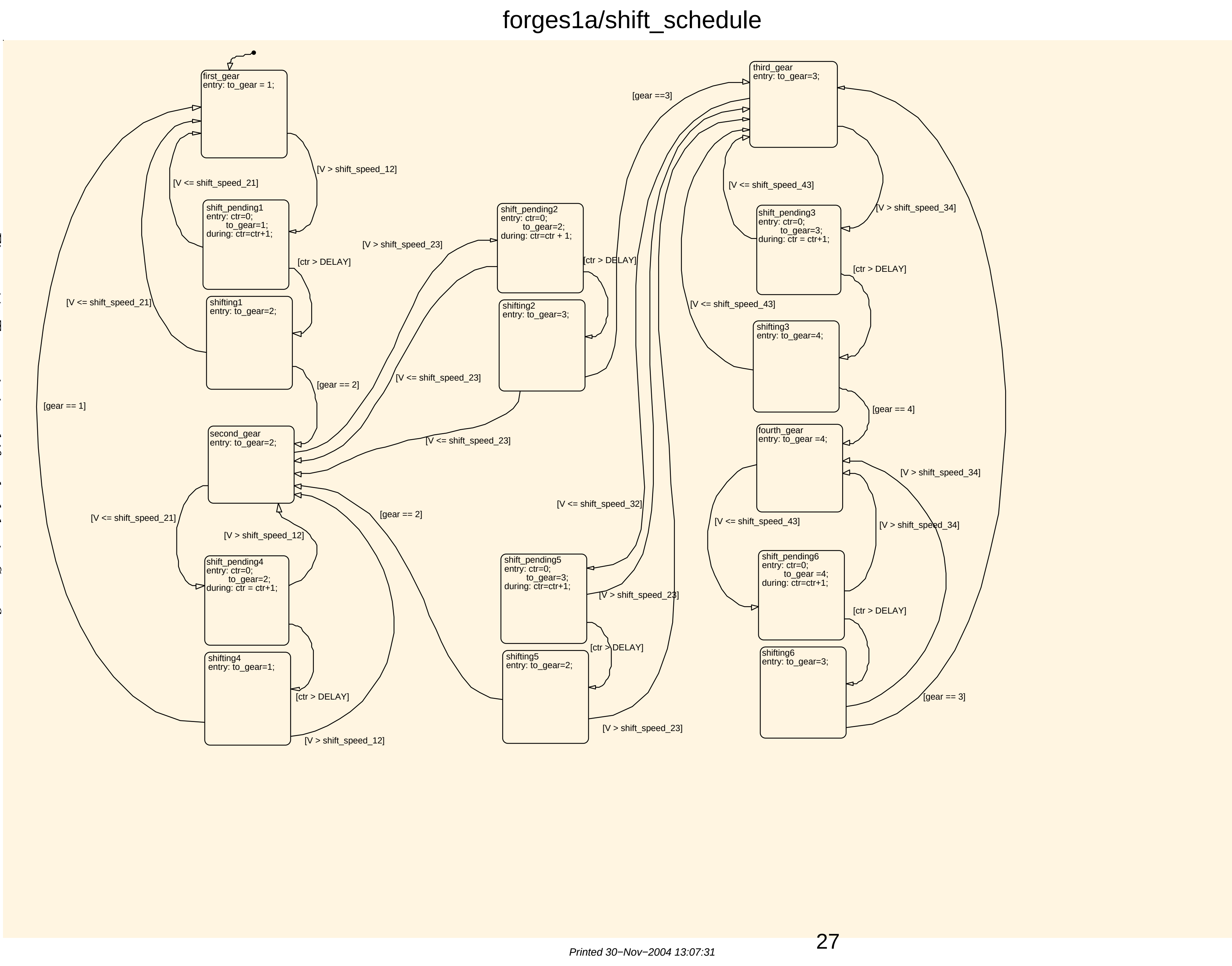


Figure 11: Transmission shift scheduler in Stateflow

```
/* Variables */

int test;
int chart_shift_schedule_2;
int state_third_gear_3;
int state_first_gear_4;
int state_shifting6_5;
int state_shifting5_6;
int state_shifting4_7;
int state_shift_pending1_8;
int state_shift_pending2_9;
int state_shift_pending3_10;
int state_shifting1_11;
int state_shifting2_12;
int state_shifting3_13;
int state_fourth_gear_14;
int state_second_gear_15;
int state_shift_pending6_16;
int state_shift_pending5_17;
int state_shift_pending4_18;
int to_gear;
int gear;
float V;
float shift_speed_12;
float shift_speed_21;
int ctr;
int DELAY;
float shift_speed_23;
float shift_speed_32;
float shift_speed_34;
float shift_speed_43;

/* Function definitions */

void forges1a () {
  if (chart_shift_schedule_2) {
    if (state_third_gear_3) {
      test = V >= shift_speed_32;
      if (test) {
        state_third_gear_3 = 0;
        state_shift_pending5_17 = 1;
        ctr = 0;
        to_gear = 3;
      } else {
        test = V > shift_speed_34;
        if (test) {
          state_third_gear_3 = 0;
          state_shift_pending3_10 = 1;
          ctr = 0;
          to_gear = 3;
        }
      }
    } else {
      if (state_first_gear_4) {
        test = V > shift_speed_12;
        if (test) {
          state_first_gear_4 = 0;
          state_shift_pending1_8 = 1;
          ctr = 0;
          to_gear = 1;
        }
      } else {
        if (state_shifting6_5) {
          test = gear == 3;
          if (test) {
            state_shifting6_5 = 0;
            state_third_gear_3 = 1;
            to_gear = 3;
          } else {
            test = V > shift_speed_34;
            if (test) {
              state_shifting6_5 = 0;
              state_fourth_gear_14 = 1;
              to_gear = 4;
            }
          }
        } else {
          if (state_shifting5_6) {
            test = V > shift_speed_23;
            if (test) {
              state_shifting5_6 = 0;
              state_third_gear_3 = 1;
              to_gear = 3;
  } else {
    test = gear == 2;
    if (test) {
      state_shifting5_6 = 0;
      state_second_gear_15 = 1;
      to_gear = 2;
    }
  }
} else {
  if (state_shifting4_7) {
    test = gear == 1;
    if (test) {
      state_shifting4_7 = 0;
      state_first_gear_4 = 1;
      to_gear = 1;
    } else {
      test = V > shift_speed_12;
      if (test) {
        state_shifting4_7 = 0;
        state_second_gear_15 = 1;
        to_gear = 2;
      }
    }
  } else {
    if (state_shift_pending1_8) {
      test = V >= shift_speed_21;
      if (test) {
        state_shift_pending1_8 = 0;
        state_first_gear_4 = 1;
        to_gear = 1;
      } else {
        test = ctr > DELAY;
        if (test) {
          state_shift_pending1_8 = 0;
          state_shifting1_11 = 1;
          to_gear = 2;
        } else {
          ctr = ctr + 1;
        }
      }
    } else {
      if (state_shift_pending2_9) {
```

```
  test = ctr > DELAY;
  if (test) {
    state_shift_pending2_9 = 0;
    state_shifting2_12 = 1;
    to_gear = 3;
  } else {
    test = V >= shift_speed_23;
    if (test) {
      state_shift_pending2_9 = 0;
      state_second_gear_15 = 1;
      to_gear = 2;
    } else {
      ctr = ctr + 1;
    }
  }
} else {
  if (state_shift_pending3_10) {
    test = V >= shift_speed_43;
    if (test) {
      state_shift_pending3_10 = 0;
      state_third_gear_3 = 1;
      to_gear = 3;
    } else {
      test = ctr > DELAY;
      if (test) {
        state_shift_pending3_10 = 0;
        state_shifting3_13 = 1;
        to_gear = 4;
      } else {
        ctr = ctr + 1;
      }
    }
  } else {
    if (state_shifting1_11) {
      test = gear == 2;
      if (test) {
        state_shifting1_11 = 0;
        state_second_gear_15 = 1;
        to_gear = 2;
      } else {
        test = V >= shift_speed_21;
        if (test) {
```

```
        state_shifting1_11 = 0;
        state_first_gear_4 = 1;
        to_gear = 1;
      }
    }
  } else {
    if (state_shifting2_12) {
      test = gear == 3;
      if (test) {
        state_shifting2_12 = 0;
        state_third_gear_3 = 1;
        to_gear = 3;
      } else {
        test = V >= shift_speed_23;
        if (test) {
          state_shifting2_12 = 0;
          state_second_gear_15 = 1;
          to_gear = 2;
        }
      }
    } else {
      if (state_shifting3_13) {
        test = gear == 4;
        if (test) {
          state_shifting3_13 = 0;
          state_fourth_gear_14 = 1;
          to_gear = 4;
        } else {
          test = V >= shift_speed_43;
          if (test) {
            state_shifting3_13 = 0;
            state_third_gear_3 = 1;
            to_gear = 3;
          }
        }
      } else {
        if (state_fourth_gear_14) {
          test = V >= shift_speed_43;
          if (test) {
            state_fourth_gear_14 = 0;
            state_shift_pending6_16 = 1;
            ctr = 0;
```

```
            to_gear = 4;
          }
        } else {
          if (state_second_gear_15) {
            test = V > shift_speed_23;
            if (test) {
              state_second_gear_15 = 0;
              state_shift_pending2_9 = 1;
              ctr = 0;
              to_gear = 2;
            } else {
              test = V >= shift_speed_21;
              if (test) {
                state_second_gear_15 = 0;
                state_shift_pending4_18 = 1;
                ctr = 0;
                to_gear = 2;
              }
            }
          } else {
            if (state_shift_pending6_16) {
              test = V > shift_speed_34;
              if (test) {
                state_shift_pending6_16 = 0;
                state_fourth_gear_14 = 1;
                to_gear = 4;
              } else {
                test = ctr > DELAY;
                if (test) {
                  state_shift_pending6_16 = 0;
                  state_shifting6_5 = 1;
                  to_gear = 3;
                } else {
                  ctr = ctr + 1;
                }
              }
            } else {
              if (state_shift_pending5_17) {
                test = ctr > DELAY;
                if (test) {
                  state_shift_pending5_17 = 0;
                  state_shifting5_6 = 1;
                to_gear = 2;
              } else {
                test = V > shift_speed_23;
                if (test) {
                  state_shift_pending5_17 = 0;
                  state_third_gear_3 = 1;
                  to_gear = 3;
                } else {
                  ctr = ctr + 1;
                }
              }
            } else {
              if (state_shift_pending4_18) {
                test = ctr > DELAY;
                if (test) {
                  state_shift_pending4_18 = 0;
                  state_shifting4_7 = 1;
                  to_gear = 1;
                } else {
                  test = V > shift_speed_12;
                  if (test) {
                    state_shift_pending4_18 = 0;
                    state_second_gear_15 = 1;
                    to_gear = 2;
                  } else {
                    ctr = ctr + 1;
                  }
                }
                return ;
              } else {
                abort
                 ("Active OR State: shift_schedule had no active child");
                return ;
              }
              return ;
            }
          }
        }
      }
    }
  }
}
```

```
                  }
                }
              }
            }
          }
        }
      }
    }
    return ;
  } else {
    to_gear = 1;
    gear = 1;
    V = 1;
    shift_speed_12 = 1;
    shift_speed_21 = 1;
    DELAY = 3;
    chart_shift_schedule_2 = 1;
    state_first_gear_4 = 1;
    to_gear = 1;
    return ;
  }
  return ;
}
```

## B Matlab Example

Figure 12 is an example of Matlab code for an opblock in SwRI's E4 experiment. The lines prefixed with `KI` have been commented out by Kestrel for two reasons. First, the check for valid data has been removed to be consistent with the SwRI hand coded C++ version of the opblock. The second change is the assignment of `0.0` to `tempsum` rather than `0`. This reflects the fact that the Matlab meta-model distinguishes integers from floats and requires that all occurrences of a variable have the same type.

Figure 13 shows the abstract syntax for the OBAverageFilter after parsing. Finally, Figure 14 shows the C code generated for OBAverageFilter.

```
function outData=OBAverageFilter(inData,nFilterSize)

% Written by:   John Signorotti
%
% Syntax:       outData=OBAverageFilter(inData,nFilterSize)
%
% Originated:   March 2003
%               Southwest Research Institute
%               6220 Culebra Road
%               San Antonio, TX 78238
%
% Purpose:      Implements running averagea filter
%
% Inputs:       inData:         data to be averaged, real or complex, vector
%
% Parameters:   nFilterSize:    number of points to average over
%
% Environemnt:  None
%
% Output:       outData:        running average of input data, real or complex, vector
%                               Note that output is shorter than input by
%                               (nFilterSize-1) points
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  Developed by Southwest Research Institute
%  $%Author: $
%  $%Archive: $
%  $%Revision: $
% ************************************************************************************

% Check for valid data
% KI if (isnan(inData(1)) | isnan(nFilterSize))
% KI     outData=NaN;
% KI     return;
% KI end

% KI tempsum=0;                         % Temporary variable for holding filter outputs
% KI begin change
tempsum=0.0;
% KI end change
outData=zeros(1,length(inData)-nFilterSize);                % Storage for output data
for i=1:length(inData)-nFilterSize+1          % Loop over each Input sample
    for j=1:nFilterSize                       % Loop over range of outDataraging
        tempsum=tempsum+inData(i+j-1);        % Accumulate data
    end
    outData(i)=tempsum/nFilterSize;           % Divide by filter length to get average
% KI    tempsum=0;                                % Reset temporary varialbe
% KI begin change
    tempsum=0.0;
% KI end change
end
return;
```

Figure 12: SwRI OpBlock for OBAverageFilter

```
psl {

  import /Matlab/Oscar/MatlabToC/Matlab

  op target : String
  def target = "AverageFilter"

  op targetFileName : String
  def targetFileName = target ++ ".h"

  op varDecls : VarDecls
  def varDecls = []

  op includes : List String
  def includes = ["forges.h", "../OPBlockCores/" ++ targetFileName]

  % op typeInstances : List (String * String * String * List (String * CGen.Type * Addr * RefType))
  def typeInstances = [
     ("main", target ++ "_Double", target, [
       ("inData", Const(Ptr Double), AutoAddr(Ptr(String "inData")),
           Buffer(Real, Ref(String "numElements", []))),
       ("numElements", Const UnsignedInt, AutoAddr(Val(String "numElements")),
           Array (Ext(Int unsignedIntType))),
       ("nFilterSize", Const Long, AutoAddr(Val(String "nFilterSize")), Array(Ext(Int longType))),
       ("outData", Ptr Double, AutoAddr(Ptr(String "outData")), Buffer(Real,
             BinExpr(Minus, Ref(String "numElements", []), Ref(String "nFilterSize", []))))])
    ]
}
```

Figure 13: Abstract Syntax for OBAverageFilter

```
/* Include files */

#include "forges.h"

#include "../OPBlockCores/AverageFilter.h"


/* Function definitions */

template <>
bool AverageFilter
    (const double* inData,
     const unsigned numElements,
     const long nFilterSize,
     double* outData) {
  printf ("AverageFilter_Double\n");
  *(((double*) sp)) = 0.0;
  zeros (outData, 1, numElements - nFilterSize, numElements - nFilterSize);
  *(((int*) sp + 8)) = 1;
  if ((*(((int*) sp + 8))) <= ((numElements - nFilterSize) + 1)) {
    do {
      *(((int*) sp + 12)) = 1;
      while ((*(((int*) sp + 12))) <= nFilterSize) {
        *(((double*) sp)) =
        (*(((double*) sp))) +
        (*(inData + (((*(((int*) sp + 8))) + (*(((int*) sp + 12)))) - 2)));
        *(((int*) sp + 12)) = (*(((int*) sp + 12))) + 1;
      }
      *(outData + ((*(((int*) sp + 8))) - 1)) =
      (*(((double*) sp))) / nFilterSize;
      *(((double*) sp)) = 0.0;
      *(((int*) sp + 8)) = (*(((int*) sp + 8))) + 1;
    } while ((*(((int*) sp + 8))) <= ((numElements - nFilterSize) + 1));
    return 1;
  }
  return 1;
}
```

Figure 14: Kestrel Generated Code for OBAverageFilter

## References

[1] Daniel Elphick, Michael Leuschel, and Simon Cox. Partial evaluation of MATLAB. In *GPCE 2003*, volume 2830 of *LNCS*, pages 344–363. Springer-Verlag, 2003.

[2] David Harel. Statecharts: A visual formalism for complex systems. *The Science of Computer Programming*, pages 231–274, 1987.

[3] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual.* Addison-Wesley, 2004.

[4] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.

[5] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Professional, 2003.

[6] The MathWorks. *Simulink Reference, Version 6.* 2004.

[7] The MathWorks. *Stateflow Reference, Version 6.* 2004.

[8] Bill Milam, 2002. Personal Communication.

[9] Mike Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2 edition, 1992.